implementing 64 bits integers in RNS (residue number system)

110 • 26 Dec 2024

Introduction

The residue number system (RNS) is a way to represent integers $\mod N$ with $N = \prod_i^k p_i$ where $\forall i, j \in [\![1, k]\!],\ i \neq j,\ \gcd(p_i, p_j) = 1$, such that some operations are much faster in the RNS than the corresponding operations in the radix representation.

The RNS allows for computations without any need to consider carries when performing additions, subtractions, or multiplications $(\mod N)$.

Moreover big numbers can be represented by using a set of much smaller numbers on which all calculations can be done at the same time. With parallelism, space is always a good trade off against time.

The size of the maximum integer N that can be represented by the RNS grows exponentially with the number of factors of N. For example, using three moduli of size $\approx 2^k$ will be able to represent numbers as big as 2^{3k}.

So it is not even a trade off of space against time. The only loss of space is due to the growing number of moduli, all of which have to be written on bigger machine numbers than their actual value.

Computing everything using the RNS looks, at first, like a good idea, and we will see it is not really the case.

Unfortunately, some operations are much more complicated in the RNS than in the radix representation, changing from the RNS to the radix representation is often needed for comparisons or divisions, and it is heavily dependent on the coprime integer used.

Yet it is fascinating to look at it, and since all the information needed is already in the residues and the Modulus, I speculate there might still be some unknown links between the two systems yet to be found that allows for faster computations, even when considering those harder operations.

Chinese Remainder Theorem

The RNS is a number representation system based on the Chinese Remainder Theorem (CRT) .

The CRT states that if you know each remainder from the Euclidian division of a number n by a set of pairwise coprimes values p_i,

$$r_i = n - \left\lfloor \frac{n}{p_i} \right\rfloor p_i$$

the remainder

$$r_n = n - \left\lfloor \frac{n}{\prod p_i} \right\rfloor \prod_i^k p_i$$

can be uniquely determined.

We define $\prod_i^k p_i = N$ for the rest of the text.

Here we use the term remainder as $r_i \in [\![0, p_i)$ but we could take any number in a range of the same size.

The RNS defines a number only by the set of "residues" $r_1, r_2, ..., r_k$. Each value between 0 and N is uniquely defined by those residues. The reconstruction of the number in a radix representation is then seen only as a conversion operation. It is easier to convert from a radix representation to the RNS than the other way around.

Here is a quick example to show how the RNS works. We choose three moduli $p_1 = 3, p_2 = 5, p_3 = 7$ as a set of coprime integers.

Their product defines $N = 3 \times 5 \times 7 = 105$.

The RNS can now uniquely define a number in $[\![0, 105)$

Let's pick up a random number $x \in [\![0, 105)$. I threw in the air a 105 balanced dice and it landed on the face 64, so $x = 64$.

$$\begin{aligned} 64 &= 3 \times 21 + 1 \equiv 1 \mod 3 \\ 64 &= 5 \times 12 + 4 \equiv 4 \mod 5 \\ 64 &= 7 \times 9 + 1 \equiv 1 \mod 7 \end{aligned} \tag{1}$$

The number 64 is uniquely determined knowing only those 3 values and that it is smaller than 105.

Reconstruction

On the other side, from those 3 residues, the number 64 can be reconstructed. How?

Let's take x as an unkown which verifies all the congruence relations that were just described. We know that $x \equiv 1 \mod 3$, so it might be possible to write x as

$$x = v + 3k$$

where $v \equiv 1 \mod 3$.

The same is true for other moduli, we might also be abe to write $x = v' + 5k'$ and $x = v'' + 5k''$ as well.

The congruence $v \equiv 1 \mod 3$ does not mean v has to be equal to one, it could be equal to a multiple of 5 and 7. Because 5 and 7 are coprime with 3 their inverse exists $\mod 3$.

If we take

$$v = e \times 5 \times 7$$

with

$$e = (1 \times 5^{-1}7^{-1}) \mod 3$$

then

$$v \equiv 1 \mod 3$$

.

The same is true for v' and v''. We can write v' as

$$v' = (4 \times 3^{-1}7^{-1})_{\mod 5} \times 3 \times 7$$

then

$$v' \equiv 4 \mod 5$$

.

and

$$v'' = (1 \times 3^{-1}5^{-1})_{\mod 7} \times 3 \times 5$$

then

$$v'' \equiv 1 \mod 7$$

.

The advantage of having

$$\begin{aligned} v &= K_1 \times 5 \times 7 \\ v' &= K_2 \times 3 \times 7 \\ v'' &= K_3 \times 3 \times 5 \end{aligned}$$

is that, then

$$\begin{aligned} v &\equiv 1 \mod 3 \\ v &\equiv 0 \mod 5 \\ v &\equiv 0 \mod 7 \end{aligned} \qquad \begin{aligned} v' &\equiv 0 \mod 3 \\ v' &\equiv 4 \mod 5 \\ v' &\equiv 0 \mod 7 \end{aligned} \qquad \begin{aligned} v'' &\equiv 0 \mod 3 \\ v'' &\equiv 0 \mod 5 \\ v'' &\equiv 1 \mod 7 \end{aligned}$$

Such that defining $X = v + v' + v''$, we now have

$$\begin{aligned} X &\equiv 1 \mod 3 \\ X &\equiv 4 \mod 5 \\ X &\equiv 1 \mod 7 \end{aligned}$$

And so,

$$X \equiv 64 \mod N \Leftrightarrow X \equiv x \mod N$$

Since $5 \equiv 2 \mod 3$ and $7 \equiv 1 \mod 3$ their inverse can be computed easily: $5^{-1} \equiv 2 \mod 3$ and $7^{-1} \equiv 1 \mod 3$.

They are their own inverse, and it's quite normal because over $\mathbb{Z}/3\mathbb{Z}$ all values n not congruent to 0 are either $n \equiv 1$ or $n \equiv -1$, so that $n^2 \equiv 1$.

So we can already write

$$X = 70 + 3k$$

such that

$$\begin{aligned} X &\equiv 1 \mod 3 \\ X &\equiv 3k \mod 5 \\ X &\equiv 3k \mod 7 \end{aligned}$$

Now with the same type of reasonning we can find that if

$$\begin{aligned} 3k &= 3 \times 7 \times 3^{-1}{}_{\mod 5} \times 7^{-1}{}_{\mod 5} \times 4 + 3 \times 5 \times 3^{-1}{}_{\mod 7} \times 5^{-1}{}_{\mod 7} \times 1 \\ &= 3 \times 7 \times 2 \times 3 \times 4 + 3 \times 5 \times 5 \times 3 \times 1 \\ &= 3 \times 7 \times 2 \times 3 \times 4 + 3 \times 5 \times 5 \times 3 \times 1 \\ &= 504 + 225 \\ &= 729 \end{aligned}$$

So

$X = 70 + 729 = 799 = 105 \times 7 + 64$ verifies (1). X is not exactly equal to 64 but it is congruent to 64 modulo 105.

If we had taken $x = 65$ instead of $x = 64$, the reconstruction would likely have been completely different and using the CRT the set of residues would have been

$$\begin{aligned} x &\equiv 1 + 1 \equiv 2 \mod 3 \\ x &\equiv 4 + 1 \equiv 0 \mod 5 \\ x &\equiv 1 + 1 \equiv 2 \mod 7 \end{aligned} \tag{2}$$

Applying the same procedure than the previous one for 64 leads to

$$X = 590 = 105 \times 5 + 65$$

The number given by the reconstruction is actually smaller for 65 than for 64 and at first it looks impossible from the residues alone to be able to tell which one is bigger.

It is also possible to reconstruct the number by parsing a lookup table containing all residues. This is basically the method we will discuss later that allows to compare two numbers represented by the RNS.

In conclusion, given a number x whose RNS decomposition is $(r_1, ..., r_k)$, the general formula to find a number in a radix representation is

$$x = \sum_{i=1}^{k} (r_i \left(\frac{N}{p_i} \right)^{-1}) \mod p_i \frac{N}{p_i} \mod N$$

but other methods are possible.

Basic operations

Addition, multiplication and subtraction $\mod N$ can be performed very easily using the RNS. The modular arithmetic tells us that you only need to perform the same operations on the residues.

For example, performing an addition between two numbers in the RNS system can be achieved by adding all the residues. Abusing notations, this translates to:

$$n_1 = (r_1, r_2, r_3)$$

$$n_2 = (r'_1, r'_2, r'_3)$$

$$n_1 + n_2 \mod N = (r + r'_1 \mod p_1, r_2 + r'_2 \mod p_2, r_3 + r'_3 \mod p_3)$$

the same is true for the multiplication and the subtraction

$$n_1 \times n_2 \mod N = (r \times r'_1 \mod p_1, r_2 \times r'_2 \mod p_2, r_3 \times r'_3 \mod p_3)$$

$$n_1 - n_2 \mod N = (r - r'_1 \mod p_1, r_2 - r'_2 \mod p_2, r_3 - r'_3 \mod p_3)$$

Computer scientists dream would be to have a $N = 2^k$because machine numbers are powers of two. If N was a power of two, the conversion between RNS and radix representation woud be super easy. But N being the product of coprime numbers, it is obviously impossible.

Let's write the first algoritms for those operations.

First, we need to set the residue number system. We want to emulate a 64 bits unsigned integer using a set of 32 bits unsigned integers, and we want to be able to check whether an addition or a subtraction overflowed N. We do not consider yet checking an overflow for multiplications here because it would require either to have N bigger than a 128 bits integer or some additionnal work.

We chose a set of 5 prime integers all around $N^{1/5}$ represented by 32 its unsigned integers, so that any product of two residue does not overflow the 32 bits. If they are prime, then they are necessarily coprime integers (as well as their product) and because they are all odd, it allows us to perform what seems complicated at first, comparison and checking the overflow during an addition or a subtraction.

We define the set of coprime integers:

```
const P1:u32 = 7121; //first modulus
const P2:u32 = 7127; //second modulus
const P3:u32 = 7129; //third modulus
const P4:u32 = 7151; //fourth modulus
const P5:u32 = 7159; //fifth modulus
const N:u128 = (P1 as u128)*(P2 as u128)*(P3 as u128)*(P4 as u128)*(P5 as u12
        8);
```

N is slightly bigger than $M = 2^{64}$. For now, operations are not computed modulo M but are computed modulo the native modulus N.

We define a struct that represents the number in the RNS:

```
#[derive(Copy, Clone)]
pub struct DoubleInteger {
    r_1: u32,
    r_2: u32,
    r_3: u32,
    r_4: u32,
    r_5: u32,

}

//constructors

impl DoubleInteger{
    pub fn new(r_1: u32,r_2: u32,r_3: u32,r_4: u32,r_5: u32) -> Self {
        Self { r_1,r_2,r_3,r_4,r_5 }
    }
}

impl DoubleInteger{
    pub fn new_from_double_integer(x:DoubleInteger) -> Self {
        let r_1=x.r_1;
        let r_2=x.r_2;
        let r_3=x.r_3;
        let r_4=x.r_4;
        let r_5=x.r_5;
        Self { r_1,r_2,r_3,r_4,r_5 }
    }
}

impl DoubleInteger{
    pub fn new_from_u64(x:u64) -> Self {
        let u =decompose(x);
        let r_1=u.r_1;
        let r_2=u.r_2;
        let r_3=u.r_3;
        let r_4=u.r_4;
        let r_5=u.r_5;
        Self { r_1,r_2,r_3,r_4,r_5 }
```

```
    }
}

impl DoubleInteger{
    pub fn assign(r_1: u32,r_2: u32,r_3: u32,r_4: u32,r_5: u32) -> Self {
        let r_1=r_1;
        let r_2=r_2;
        let r_3=r_3;
        let r_4=r_4;
        let r_5=r_5;

        Self { r_1,r_2,r_3,r_4,r_5 }
    }
}
```

The addition function can easily be written:

```
pub fn add(x:& DoubleInteger,y:& DoubleInteger)->DoubleInteger
{
    let r_1=(x.r_1+y.r_1)%P1;
    let r_2=(x.r_2+y.r_2)%P2;
    let r_3=(x.r_3+y.r_3)%P3;
    let r_4=(x.r_4+y.r_4)%P4;
    let r_5=(x.r_5+y.r_5)%P5;


    let mut temp_sum = DoubleInteger::new(r_1,r_2,r_3,r_4,r_5);

    temp_sum
}
```

The multiplication can be written

```
pub fn mul(x:& DoubleInteger,y:& DoubleInteger)->DoubleInteger
{
    let r_1=(x.r_1*y.r_1)%P1;
    let r_2=(x.r_2*y.r_2)%P2;
    let r_3=(x.r_3*y.r_3)%P3;
    let r_4=(x.r_4*y.r_4)%P4;
    let r_5=(x.r_5*y.r_5)%P5;
```

```
    let mut temp_prod = DoubleInteger::new(r_1,r_2,r_3,r_4,r_5);

    temp_prod
}
```

and the subtraction

```
pub fn sub(x:& DoubleInteger,y:& DoubleInteger)->DoubleInteger
{
    let r_1=((x.r_1+P1)-y.r_1)%P1;
    let r_2=((x.r_2+P2)-y.r_2)%P2;
    let r_3=((x.r_3+P3)-y.r_3)%P3;
    let r_4=((x.r_4+P4)-y.r_4)%P4;
    let r_5=((x.r_5+P5)-y.r_5)%P5;


    let mut temp_sub = DoubleInteger::new(r_1,r_2,r_3,r_4,r_5);
    temp_sub
}
```

Here, we added P_i to each subtraction so that it is never negative. We used unsigned 32 bits integers that would not allow for negative values.

Now we need to reconstruct the number. In order to do that we can use immediately the formula from the CRT or we can use a mixed radix representation. We will discuss the second method later.

```
const P1_128:u128 = 7121;
const P2_128:u128 = 7127;
const P3_128:u128 = 7129;
const P4_128:u128 = 7151;
const P5_128:u128 = 7159;

const PROD3451_128:u128 = P1_128*P3_128*P4_128*P5_128;
const PROD3452_128:u128 = P2_128*P3_128*P4_128*P5_128;
const PROD1254_128:u128 = P1_128*P2_128*P4_128*P5_128;
const PROD1253_128:u128 = P1_128*P2_128*P3_128*P5_128;
const PROD1234_128:u128 = P1_128*P2_128*P3_128*P4_128;

pub fn reconstruct(x:&DoubleInteger)->u128{
```

```
    let fact_1 =((x.r_1 as u128)*INV1 as u128)%P1_128;
    let fact_2 =((x.r_2 as u128)*INV2 as u128)%P2_128;
    let fact_3 =((x.r_3 as u128)*INV3 as u128)%P3_128;
    let fact_4 =((x.r_4 as u128)*INV4 as u128)%P4_128;
    let fact_5 =((x.r_5 as u128)*INV5 as u128)%P5_128;

    let mut r= PROD3452_128*fact_1+PROD3451_128*fact_2 + PROD1254_128*fact_3
        + PROD1253_128*fact_4 + PROD1234_128*fact_5;
    r=r%N;


    return r as u128;
}
```

Defining $N_i = \frac{N}{p_i}$, we have

$$\begin{aligned}
\text{INV1} &\equiv N_1^{-1} \mod p_1 \\
\text{INV2} &\equiv N_2^{-1} \mod p_2 \\
\text{INV3} &\equiv N_3^{-1} \mod p_3 \\
\text{INV4} &\equiv N_4^{-1} \mod p_4 \\
\text{INV5} &\equiv N_5^{-1} \mod p_5
\end{aligned}$$

Rightshift and leftshift

Can we make some more sophisticated operations? Can we make a right shift and a left shift?

Depending on the machine, the system or the programming language, right and left shifts have no unique definition. We will call right shift an integer division by 2^t and left shift a *wrapping* multiplication by 2^t. This is not a "conventionnal left shift where 2^{64} should be equal to zero. For now, shifts are also $\mod N$ with the restriction that left shifts bigger than 63 are not allowed, as it would be for 64 bits unsigned integers, but since it is possible to perform several smaller leftshifts, it is possible to overflow and wrapp around 2^64. We will see a "true" left shift later.

left shift

Multiplying by two in the RNS is just like multiplying by two in the real world but because we use only 32 bits unsigned integers to emulate 64 bits or so integers, we need to separate our calculation into two cases. The case where the desired left shift is smaller than 32 bits, and the case where the left shift is bigger than 32 bits.

```
pub fn leftbitshift(x:&DoubleInteger, t:u32 )->DoubleInteger{
    if t<32
    {
        let y = DoubleInteger::new((1<<t)%P1,(1<<t)%P2,(1<<t)%P3,(1<<t)%P4,
        (1<<t)%P5);
        let r=mul(x,&y);
        return r;
    }else if t < 64
    {
        let t_1= t/2;
        let t_2 = t -t_1;

        let y = DoubleInteger::new((1<<t_1)%P1,(1<<t_1)%P2,(1<<t_1)%P3,(1<<t_
        1)%P4,(1<<t_1)%P5);
        let z = DoubleInteger::new((1<<t_2)%P1,(1<<t_2)%P2,(1<<t_2)%P3,(1<<t_
        2)%P4,(1<<t_2)%P5);

        let mut r=mul(x,&y);
        r=mul(&r,&z);
        return r;
    }else
    {
        panic!("shift is too big!");
    }

}
```

right shift

The right shift is a bigger piece. We need to know the parity of the number. Getting the parity of a number is simple if the RNS uses 2^k as a coprime integer. But taking 2^k as a coprime integer does not allow for a right shift because then 2 is no longer invertible.

The basic algorithm to make a 1 bit right shift would be:

- find the parity of the number
- if the parity is 0, multiply by the inverse of 2,
- if the parity is 1, subtract 1 and then multiply by the inverse of 2

inverse of 2

Multiplying by the inverse of 2 is exactly like dividing by 2 if the number is a multiple of two. The same observation will allow for integer divisions for other numbers as well, but it is a much longer story.

It is quite easy to find the inverse of $2 \mod N$. We can proove that $2^{-1} \equiv \frac{N+1}{2} \mod N$. Which means we can multiply by the inverse of 2 very quickly. Since N is odd,

$$N + 1 = 2m + 2 = 2(m + 1)$$

thus

$$\begin{aligned} N + 1 &\equiv 1 \mod N \\ &\Leftrightarrow \\ (m + 1) &\equiv 2^{-1} \mod N \\ &\Leftrightarrow \\ (m + 1) &\equiv \frac{N + 1}{2} \mod N \\ &\Leftrightarrow \\ 2^{-1} &\equiv \frac{N + 1}{2} \mod N \end{aligned}$$

As we stated before multiplying by the inverse of 2 an even number is exactly like dividing y two. For example

$$2^{-1} \equiv 8 \mod 15$$

$$8 \times 6 \equiv 48 \equiv 3 \times 15 + 3 \equiv 3 = \frac{6}{2} \mod 15$$

Multiplying by the inverse of 2 odd numbers is not really more complicated

$$8 \times 7 \equiv 8 \times (6 + 1) \equiv 8 \times 6 + 8 \times 1 \equiv 3 + 8 \equiv 11 \mod 15$$

This will be used in the rightshift algorithm when each residue has to be multiplied by the inverse of 2. If the residue of a number $\mod p_i$ is even we will apply a simple rightshift to it, otherwise, we apply a rightshift and add $\frac{p_i+1}{2}$:

```
r_i = if r_i&1 == 0 { r_i>>1 } else { (r_i>>1) + INV_2_Pi };
```

finding parity of a number

The RNS does not allow immediatly to tell whether a number is odd or even if we did not include a power of two in our coprime system. On the other side we could easily find the parity of a number knowing its radix representation.

As mentionned earlier there are several ways to change from the RNS to the radix representation. One way to do so, tedious at first sight, is by parsing a lookup table that stores all the RNS possible values in the range $[\![0, N)$. Let's go back to our simple example with $N = 3 \times 5 \times 7$. We present the beginning of that lookup table where n is the radix representation of the number, and p is the parity of n:

$p_1 = 3$	$p_2 = 5$	$p_3 = 7$	**n**	**p**
r_1	r_2	r_3		
0	0	0	0	0
1	1	1	1	1
2	2	2	2	0
0	3	3	3	1
1	4	4	4	0
2	0	5	5	1
0	1	6	6	0
1	2	0	7	1
2	3	1	8	0
0	4	2	9	1

Up until $n = 6$ the parity of n is the parity of $r_3 \equiv n \mod 7$. For $7 \leqslant n < 14$, the parity of n is the opposite of the parity of r_3. We also know that the residue of $n \mod 7$ will always be any value between 0 and 6 included.

Thus to reconstruct a number and find its parity, one ossibility is to:

- Define a number n_{temp} that starts at $r_3 \equiv n \mod 7$ in the lookup table. If 7 is added to that value, n_{temp} will always verify $n_{temp} \equiv n \mod 7$.

- repeatedly add 7 to n_{temp}, (which means jumping by 7 rows in the lookup table), until $r_2 = n_{temp} \mod 5 = n \mod 5$. The number of times 7 is defined as k_1. Each time 7 is added the parity of n_{temp} changes, so the parity of n_{temp} is now the parity of $r_3 + k_1$.
- Once $r_3 \equiv n \mod 7$ and $r_2 \equiv n \mod 5$, add repeatedly $p_3 \times p_2 = 35$ to n_{temp} until $r_1 \equiv n \mod 3$. We call the number of times 35 is added, k_2.

Because 7 is coprime with 5, $0 \leqslant k_1 \leqslant 4$.

We have two results here.

First the parity of n is the parity of $r_3 + k_1 + k_2$:

$$p \equiv r_3 + k_1 + k_2 \mod 2$$

Secondly,

$$n = r_3 + k_1 \times 7 + k_2 \times 7 \times 5.$$

The second result is a mixed radix representation of n and does not necessitate the reduction $\mod N$.

The method can be generalised to more than 3 moduli.

We can actually deduce more than that.

Because

$$n = r_3 + k_1 p_3 + k_2 p_3 p_2$$

we get that

$$n \equiv r_3 + k_1 p_3 + k_2 p_3 p_2 \mod m$$

for any number, thus, let's say m is a 32 bits integer, we can find all residues $r_{im} \equiv (n \mod m) \mod p_i$ by computing:

$$r_{im} \equiv (r_3 + k_1 p_3 + k_2 p_3 p_2 \mod m) \mod p_i.$$

We already know that

$$0 \leqslant k_i < p_i.$$

In addition

$$p_3 p_2 \mod m$$

and

$$p_3 \mod m$$

can be precomputed. Since we chose $p_i < \sqrt{2^{32}}$, the calculation

$$k_2 p_3 p_2 \mod m \mod p_i$$

as well as

$$k_1 p_3 \mod m \mod p_i$$

can not overflow the unsigned integer over 32 bits.

Therefore, to get the integer division of a number n by m the procedure will be similar to dividing by two.

- find the RNS representation of the residue $r \equiv n \mod m$
- subtract r to n in the RNS
- multiply by the inverse of m if that inverse exists

The inverse of m is uniquely defined only for integers m not being multiples of any p_i, so the fewer of them there are, the better.

how to find the ks?

The question is now to find those k_i without having access to an actual lookup table storing all the values up to N? It's easy.

We starts with $r_3 = n \mod p_3$, and at that point $n_{temp} \equiv r_2 \mod p_2$. But because $p_3 > p_2$, $r_2 = r_3$.

We write $n \mod p_i$ as n_i We want to find k_1 such that

$$\begin{aligned} r_2 + k_1 p_3 &\equiv n_2 \mod p_2 \\ &\Leftrightarrow \\ r_3 + k_1 p_3 &\equiv n_2 \mod p_2 \\ &\Leftrightarrow \\ k_1 &\equiv (n_2 - r_3) p_3^{-1} \mod p_2 \end{aligned}$$

A similar procedure is applied for k_2. Starting again from r_3

$$\begin{aligned} r_3 + k_1 p_3 + k_2 p_2 p_3 &\equiv n_1 \mod p_1 \\ &\Leftrightarrow \\ k_2 &\equiv (p_2 p_3)^{-1} (n_1 - r_3 + k_1 p_3) \mod p_1 \end{aligned}$$

Ok so now, the code to find the parity of a number in the RNS

```
const INVP5MODP4:u32=894;
const INVP5P4MODP3:u32=1415;
const INVP5P4P3MODP2:u32=116;
const INVP5P4P3P2MODP1:u32=510;

const P5_1:u32=P5%P1;
const P5_2:u32=P5%P2;
const P5_3:u32=P5%P3;

const P54_1:u32= (P5_1*P4_1)%P1;
const P54_2:u32= (P5_2*P4_2)%P2;

const P543_1:u32= (P54_1*P3_1)%P1;


const PRODUCT3:u32 = (INVP5P4MODP3*(P3-P5_3))%P3;
const PRODUCT2_1:u32 = (INVP5P4P3MODP2*(P2-P5_2))%P2;
const PRODUCT2_2:u32 = (INVP5P4P3MODP2*(P2-P54_2))%P2;
const PRODUCT1_1:u32 = (INVP5P4P3P2MODP1*(P1-P5_1))%P1;
const PRODUCT1_2:u32 = (INVP5P4P3P2MODP1*(P1-P54_1))%P1;
const PRODUCT1_3:u32 = (INVP5P4P3P2MODP1*(P1-P543_1))%P1;

const P4_D:u32 = P4<<1;
const P3_D:u32 = P3<<1;
const P2_D:u32 = P2<<1;
const P1_D:u32 = P1<<1;

pub fn parity_find( x: &DoubleInteger )->(u32,u32,u32,u32,u32)
{
    let k_1 = (INVP5MODP4*(P4_D + x.r_4 - x.r_5))%P4;
    let k_2 = (INVP5P4MODP3*(P3_D+ x.r_3 - x.r_5 )+k_1*PRODUCT3)%P3;
    let k_3 = (INVP5P4P3MODP2*( P2_D + x.r_2 - x.r_5 )+ k_1*PRODUCT2_1+k_2*PR
        ODUCT2_2 )%P2;
    let k_4 = (INVP5P4P3P2MODP1*(P1_D + x.r_1 -x.r_5 ) + k_1*PRODUCT1_1+k_2*P
        RODUCT1_2 + k_3*PRODUCT1_3 )%P1;

    let parity= (x.r_5+k_1+k_2+k_3+k_4)&1;
```

```
    (parity,k_1,k_2,k_3,k_4)
}
```

And we can now create a new reconstruction function:

```
pub fn reconstruct_2(x:&DoubleInteger)->u128{

    let parity = parity_find(x);
    let k_1 = parity.1 as u128;
    let k_2 = parity.2 as u128;
    let k_3 = parity.3 as u128;
    let k_4 = parity.4 as u128;


    let r = x.r_5 as u128 + ( P5_128*(k_1 + P4_128*(k_2 + P3_128*(k_3 + k_4*P
        2_128) ) ) ) ;

    return r as u128;
}
```

From the benchmark I run, this method is surprisingly not really faster than the first reconstruction method.

```
reconstruct 118446744073709551615:18446744073709551615
Elapsed time for reconstruction 1: 1.05µs
reconstruct 118446744073709551615:18446744073709551615
Elapsed time for reconstruction 2: 3.637µs
reconstruct 118446744073709551613:18446744073709551613
Elapsed time for reconstruction 1: 775ns
reconstruct 218446744073709551613:18446744073709551613
Elapsed time for reconstruction 2: 875ns
```

64 bits integer

Up until now, all operations were performed over the modulus $N = \prod_i^k p_i$, but we want to simulate 64 bits integers.

addition, subtraction, multiplication

The addition algorithm takes two numbers $x, y \in [\![0, 2^{64})$ and add them modulo 2^{64}. Finding the value $\mod 2^{64}$ while the native modulus is N is a bit tedious. If we were to look at that sum in $\mathbb{N}$, the addition would result in:

$$\begin{aligned} 0 \leqslant x < 2^{64} \\ 0 \leqslant y < 2^{64} \\ 0 \leqslant x + y < 2^{65} \end{aligned}$$

but

$$2^{64} < N < 2^{65}$$

parity

With the parity function we can actually determine whether

- $N \leqslant x + y < 2^{65}$
- $2^{64} \leqslant x + y < N$

$$\begin{aligned} 0 \leqslant x < 2^{64} \\ 0 \leqslant y < 2^{64} \\ 0 \leqslant x + y < 2^{65} \end{aligned}$$

How can we do that?

If x and y have the same parity (either both are odd or both are even), then $x + y$ is even. Because N is an odd number, if $x + y$ is even in $\mathbb{N}$ then $x + y \mod N$ should be even if $x + y < N$ and it should be odd if $x + y \geqslant N$.

The same is true if $x + y$ is odd. If $x + y \in \mathbb{N}$ is odd, then $x + y \mod N$ is odd if $x + y < N$ and $x + y \mod N$ is even if $x + y \geqslant N$.

Now, something similar happens when subtracting. If $x \geqslant y$, $x - y$ keeps the same parity than $x - y \in \mathbb{N}$ but if $x < y$, $x - y \mod N$ changes parity. This allows to make comparisons between numbers $\mod N$. We can thus apply that principle to $x + y - 2^{64}$.

```
pub fn is_above_m_mod_n(x:& DoubleInteger, m: &DoubleInteger)-> bool
{
```

```
    let temp_x  = DoubleInteger::new(
    (( x.r_1 + P1) - m.r_1)%P1,
    (( x.r_2 + P2) - m.r_2)%P2,
    (( x.r_3 + P3) - m.r_3)%P3,
    (( x.r_4 + P4) - m.r_4)%P4,
    (( x.r_5 + P5) - m.r_5)%P5
    );
    let parity_x = parity_find(x);
    let parity_temp_x = parity_find(&temp_x);
    if parity_x.0 == parity_temp_x.0
    {
        return true;
    }else
    {
        return false;
    }
}

pub fn add_mod_m(x:& DoubleInteger,y:& DoubleInteger)->DoubleInteger
{

    let max = DoubleInteger::new(MAX1,MAX2,MAX3,MAX4,MAX5);
    let mut xt = DoubleInteger::new(x.r_1,x.r_2,x.r_3,x.r_4,x.r_5);
    let mut yt = DoubleInteger::new(y.r_1,y.r_2,y.r_3,y.r_4,y.r_5);

    let parity_x = parity_find(x);
    let parity_y = parity_find(y);

    let mut temp_sum = DoubleInteger::new((xt.r_1+yt.r_1)%P1,(xt.r_2+yt.r_2)
        %P2,(xt.r_3+yt.r_3)%P3,(xt.r_4+yt.r_4)%P4,(xt.r_5+yt.r_5)%P5);
    let parity_sum = parity_find(&temp_sum);
    if is_above_m_mod_n(&temp_sum,&max)
    {
        //println!("M < sum < N");
        temp_sum.r_1 = (temp_sum.r_1 + NM1)%P1;
        temp_sum.r_2 = (temp_sum.r_2 + NM2)%P2;
        temp_sum.r_3 = (temp_sum.r_3 + NM3)%P3;
        temp_sum.r_4 = (temp_sum.r_4 + NM4)%P4;
        temp_sum.r_5 = (temp_sum.r_5 + NM5)%P5;
    } else if parity_x.0 == parity_y.0
    {
        //println!(" 1 1 1");
        if parity_sum.0 == 1
```

```
        {
            temp_sum.r_1 = (temp_sum.r_1 + NM1)%P1;
            temp_sum.r_2 = (temp_sum.r_2 + NM2)%P2;
            temp_sum.r_3 = (temp_sum.r_3 + NM3)%P3;
            temp_sum.r_4 = (temp_sum.r_4 + NM4)%P4;
            temp_sum.r_5 = (temp_sum.r_5 + NM5)%P5;
        }

    }else if  parity_x.0 != parity_y.0
    {
        if parity_sum.0 == 0
        {
            //println!(" 0 1 0 || 1 0 0");
            temp_sum.r_1 = (temp_sum.r_1 + NM1)%P1;
            temp_sum.r_2 = (temp_sum.r_2 + NM2)%P2;
            temp_sum.r_3 = (temp_sum.r_3 + NM3)%P3;
            temp_sum.r_4 = (temp_sum.r_4 + NM4)%P4;
            temp_sum.r_5 = (temp_sum.r_5 + NM5)%P5;
        }

    }

    return temp_sum;


}
```

Similar problems happen for the subtraction, and the multiplication, but that post has already been too long, so here is just the code.

```
pub fn sub_mod_m(x:&DoubleInteger,y:&DoubleInteger)->DoubleInteger
{
    let r_1 = (x.r_1+P1)-y.r_1 ;
    let r_2 = (x.r_2+P2)-y.r_2 ;
    let r_3 = (x.r_3+P3)-y.r_3 ;
    let r_4 = (x.r_4+P4)-y.r_4 ;
    let r_5 = (x.r_5+P5)-y.r_5 ;

    let max = DoubleInteger::new(MAX1,MAX2,MAX3,MAX4,MAX5);
    let mut res =DoubleInteger::new(r_1,r_2,r_3,r_4,r_5);
```

```
    let parity_x = parity_find(&x).0;
    let parity_y = parity_find(&y).0;

    let parity = parity_find(&res).0;
    if parity_x==parity_y
    {
        if parity == 1
        {
            res.r_1 = (r_1+P1-NM1)%P1;
            res.r_2 = (r_2+P2-NM2)%P2;
            res.r_3 = (r_3+P3-NM3)%P3;
            res.r_4 = (r_4+P4-NM4)%P4;
            res.r_5 = (r_5+P5-NM5)%P5;

        }

    }else
    {
        if parity == 0
        {
            res.r_1 = (r_1+P1-NM1)%P1;
            res.r_2 = (r_2+P2-NM2)%P2;
            res.r_3 = (r_3+P3-NM3)%P3;
            res.r_4 = (r_4+P4-NM4)%P4;
            res.r_5 = (r_5+P5-NM5)%P5;
        }

    }
    if res.r_1 == MAX1 && res.r_2 == MAX2 && res.r_3 == MAX3 && res.r_4 ==
        MAX4 && res.r_5 == MAX5
    {
        res.r_1 = 0;
        res.r_2 = 0;
        res.r_3 = 0;
        res.r_4 = 0;
        res.r_5 = 0;
    }
    if is_above_m_mod_n(&res,&max)
    {
        res.r_1 = (res.r_1 + NM1)%P1;
        res.r_2 = (res.r_2 + NM2)%P2;
        res.r_3 = (res.r_3 + NM3)%P3;
        res.r_4 = (res.r_4 + NM4)%P4;
```

```
        res.r_5 = (res.r_5 + NM5)%P5;
    }

    return res;
}
pub fn mul_mod_m(x:&DoubleInteger,y:&DoubleInteger)->DoubleInteger{

    let max = DoubleInteger::new(MAX1,MAX2,MAX3,MAX4,MAX5);

    let mut xt = DoubleInteger::new(x.r_1,x.r_2,x.r_3,x.r_4,x.r_5);
    let mut yt = DoubleInteger::new(y.r_1,y.r_2,y.r_3,y.r_4,y.r_5);
    let xh = rightbitshift(&xt,32);
    let xl_mask = leftbitshift_mod_m(&xh,32);
    let xl = sub_mod_m(&xt,&xl_mask);

    let yh = rightbitshift(&yt,32);
    let yl_mask = leftbitshift_mod_m(&yh,32);
    let yl = sub_mod_m(&yt,&yl_mask);


    //*
    let mut xl_yh =mul(&xl,&yh);
    let mut xh_yl =mul(&xh,&yl);
    let xl_yl =mul(&xl,&yl);

    // */

    let mut sum = add_mod_m(&xl_yh,&xh_yl);
    sum = leftbitshift_mod_m(&sum,32);
    sum = add_mod_m(&sum,&xl_yl);

    sum

    // separate x and y. x = xh*2^32 + xl, y = yh*2^32 + yl
    // x*y = xh*yl*2^32 + xl*yh*2^32 + xl*yl
}
```

It’s a bit long and it provides a multiplication 22 times longer than with 64 bits integers (with 5 numbers just "twice" as small). In the end, adding a new modulus inside the natural modulus does not sounds like a good idea for efficiency. Some areas might benefit from it though.

As a bonus, a division by 3 algorithm. There should be a check for the cases where $n = 0, 1, 2$, otherwise it should work all the time.

```
const INV3_P1:u32 = 2374;
const INV3_P2:u32 = 2376;
const INV3_P3:u32 = 4753;
const INV3_P4:u32 = 2384;
const INV3_P5:u32 = 4773;

pub fn find_k_i(x: &DoubleInteger)->(u32,u32,u32,u32)
{
    let k_1 = (INVP5MODP4*(P4_D + x.r_4 - x.r_5))%P4;
    let k_2 = (INVP5P4MODP3*(P3_D+ x.r_3 - x.r_5 )+k_1*PRODUCT3)%P3;
    let k_3 = (INVP5P4P3MODP2*( P2_D + x.r_2 - x.r_5 )+ k_1*PRODUCT2_1+k_2*PR
        ODUCT2_2 )%P2;
    let k_4 = (INVP5P4P3P2MODP1*(P1_D + x.r_1 -x.r_5 ) + k_1*PRODUCT1_1+k_2*P
        RODUCT1_2 + k_3*PRODUCT1_3 )%P1;
    (k_1,k_2,k_3,k_4)
}

pub fn rest_mod_3(x:&DoubleInteger)->DoubleInteger{

    let k = find_k_i(x);
    //let r = (x.r_5%3 + P5_mod_3*(k.0%3) + P54_mod_3*(k.1%3) +
        P543_mod_3*(k.2%3) + P5432_mod_3*(k.3%3))%3;
    let r = (x.r_5 + P5_mod_3*k.0 + P54_mod_3*k.1 + P543_mod_3*k.2 +
        P5432_mod_3*k.3)%3;

    let rest = DoubleInteger::new(r,r,r,r,r);
    rest
}

pub fn divide_by_3(x:&DoubleInteger)->DoubleInteger{
    let rest = rest_mod_3(x);
    let int = sub_mod_m(x,&rest);
    let div = DoubleInteger::new((int.r_1*INV3_P1)%P1,(int.r_2*INV3_P2)%P2,(i
        nt.r_3*INV3_P3)%P3,(int.r_4*INV3_P4)%P4,(int.r_5*INV3_P5)%P5);
    div
}
```

Next post will be about a baby step giant step reversed algorithm.